v1.0
20130510

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO
a.a 2013/2014
Lecturer: Alessandro Ricci

[module lab 2.3] IMPLEMENTING MONITORS IN JAVA

IMPLEMENTING MONITORS IN JAVA

- Two basic approaches to develop monitors in Java:
  - exploiting low-level Java mechanisms (`synchronized, wait, notify, notifyAll`)
  - exploiting high-level `java.util.concurrent` support

FIRST APPROACH

- Monitors can be implemented in Java by exploiting the two basic mechanisms
  - synchronization blocks
    - the bytecode instructions for entering and exiting a synchronized block are called `monitorenter` and `monitorexit`
    - Java's builtin intrinsic locks are sometimes called monitor locks or monitors
  - set of mechanisms used for explicit synchronization among threads, through shared objects
    - **`wait`**, **`notify`**, **`notifyAll`**

EXPLICIT SYNCHRONIZATION: DETAILS

- **`wait`** method
  - any synchronized method in any object can contain a `wait`, which suspend the current thread
- **`notify`** method
  - *one* (arbitrarily chosen) thread waiting on the target object is resumed upon invocation of method notify
  - also the `notify` method must be contained in a synchronized method or block
- **`notifyAll`** method
  - *all* threads waiting on the target object are resumed upon the invocation of the method `notifyAll` on the target object
  - also the `notifyAll` method must be contained in a synchronized method or block

WAIT SEMANTICS

- A wait invocation results in the following actions:
  - If the current thread has been interrupted, then the method exits immediately, throwing an `InterruptedException`
    - Otherwise, the current thread is blocked
  - The JVM places the thread in the internal and otherwise inaccessible wait set associated with the target object
  - The synchronization lock for the target object is released, but all other locks held by the thread are retained
    - A full release is obtained even if the lock is re-entrantly held due to nested synchronized calls on the target object
    - upon later resumption, the lock status is fully restored

NOTIFY SEMANTICS

- A `notify` invocation results in the following actions:
  - If one exists, an arbitrarily chosen thread, say T, is removed by the JVM from the internal wait set associated with the target object.
    - There is no guarantee about which waiting thread will be selected when the wait set contains more than one thread
  - T must re-obtain the synchronization lock for the target object, which will always cause it to block at least until the thread calling notify releases the lock.
    - It will continue to block if some other thread obtains the lock first.
  - T is then resumed from the point of its wait
- A `notifyAll` works in the same way as notify except that the steps occur (in effect, simultaneously) for all threads in the wait set for the object.
  - however, because they must acquire the lock, threads continue one at a time

WAIT INTERRUPTION

- If `Thread.interrupt` is invoked for a thread suspended in a wait, the same notify mechanics apply, except that after re-acquiring the lock, the method throws an `InterruptedException` and the thread's interruption status is set to false.
- If an interrupt and a notify occur at about the same time, there is no guarantee about which action has precedence, so either result is possible
  - Future revisions of JLS may introduce deterministic guarantees about these outcomes.

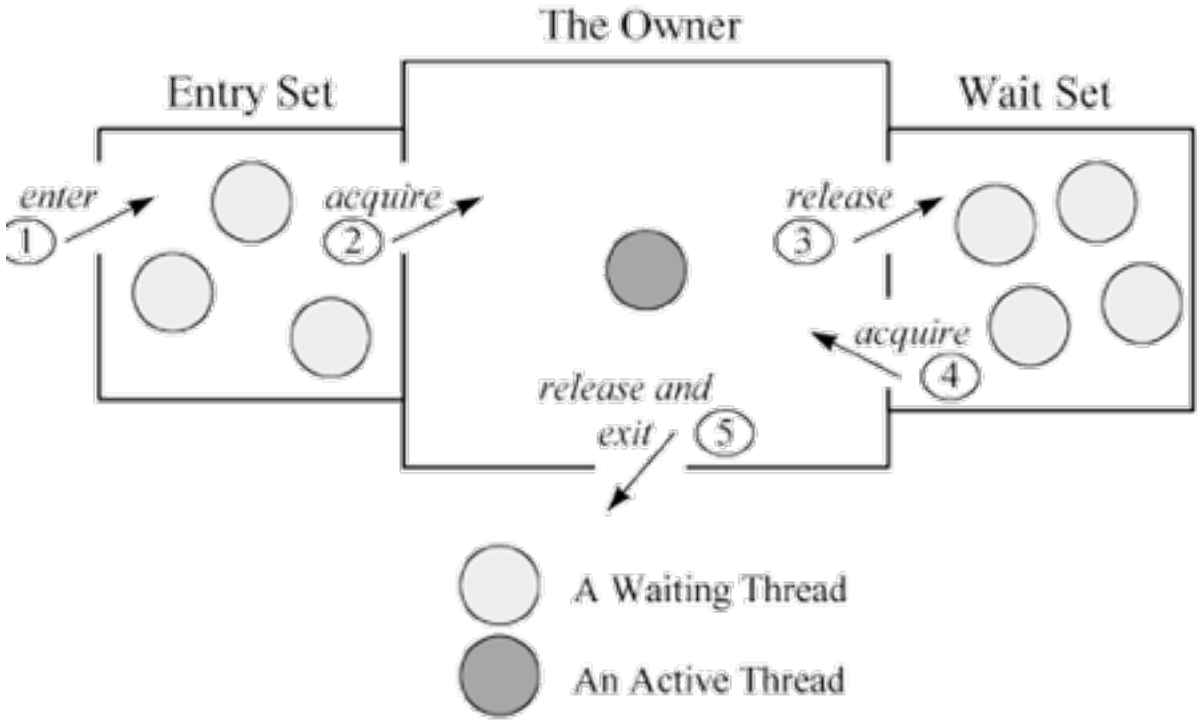
WAIT-TIMED VERSION

- The timed versions of the wait method, `wait(long msecs)` and `wait(long msecs, int nanosecs)`, take arguments specifying the desired maximum time to remain in the wait set.
- They operate in the same way as the untimed version except that if a wait has not been notified before its time bound, it is released automatically.
- There is no status indication differentiating waits that return via notifications versus time-outs.
  - Counter-intuitively, `wait(0)` and `wait(0,0)` both have the special meaning of being equivalent to an ordinary untimed `wait()`.
- A timed wait may resume an arbitrary amount of time after the requested bound due to thread contention, scheduling policies, and timer granularities.
  - There is no guarantee about granularity. Most JVM implementations have observed response times in the 1-20ms range for arguments less than 1ms.

MONITOR PATTERN

- An object following the monitor pattern encapsulates all its mutable state and guards it with object's own intrinsic lock
- Rules
  - *every* public method must be implemented as synchronized
  - no public field
  - monitor code must access / use only objects completely confined inside the monitor
  - a single condition variable is available, which is the object itself
    - wait, notify,notifyAll as waitC and signalC
- Signaling semantics: variant of Signal-and-Continue strategy
  - **E = W < S**

ENTRY AND WAIT SET



- ***Entry* set**
  - set where threads waiting for the lock are suspended
- ***Wait* set**
  - set where threads that executed a wait are waiting to be notified

EXAMPLE #1

- Simple counter monitor

```
public class Counter {

  private int count;

  public Counter(){
    count = 0;
  }

  public synchronized void inc(){
    count++;
  }

  public synchronized int getValue(){
    return count;
  }
}
```

EXAMPLE #2

- Synchronized cell

```
public class SynchCell {

  private int value;
  private boolean available;

  public SynchCell(){
    available = false;
  }

  public synchronized void set(int v){
    value = v;
    available = true;
    notifyAll();
  }

  public synchronized int get() {
    while (!available){
      try {
        wait();
      } catch (InterruptedException ex){}
    }
    return value;
  }
}
```

REMARKS

- Limits of the Java basic support
  - multiple condition predicates must be associated to the same (unique) condition variable
    - multiple threads with different roles waiting for different condition predicates can be waiting on the same (implicit) condition variable
  - `wait` semantics include “spurious wake up” (check Java doc)
    - not in response to any thread calling `notify`

> Consequences
  - to awake the desired threads, all the threads waiting on the condition variable must be awakened
  - a thread waiting on the cond variable can be awakened even if its specific condition predicate is not satisfied

> Basic “safe” implementation schema
  - wrapping `wait` in **while loop** checking the specific condition predicate
  - using **`notifyAll`** instead of `notify`

EXAMPLE #3

```
public class SynchAdder {
  private int x, y;
  boolean xAvailable, yAvailable;

  public SynchAdder(){
    xAvailable = yAvailable = false;
  }
  public synchronized void setFirstOperand(int x){
    while (xAvailable) {
      wait();
    }
    this.x = x; xAvailable = true;
    if (xAvailable && yAvailable){
      notifyAll();
    }
  }
  public synchronized void setSecondOperand(int y){
    while (yAvailable) {
      wait();
    }
    this.y = y; yAvailable = true;
    if (xAvailable && yAvailable){
      notifyAll();
    }
  }
  public synchronized int getSum() throws InterruptedException {
    while (!(xAvailable && yAvailable)){
      wait();
    }
    xAvailable = yAvailable = false;
    notifyAll();
    return x + y;
  }
}
```

- Getting the sum of the two operands only when both operands are available
  - computing + synchronizing functionality
- Multiple threads waiting on different cond predicates on the same cond variable
  - using `notifyAll`
  - using a loop for predicate check and `wait`

IMPLEMENTING A BOUNDED-BUFFER

```
public class BoundedBuffer<Item>  {
    private int first;
    private int last;
    private int count;
    private Item[] buffer;

    public BoundedBuffer(int size){
        first = 0;
        last = 0;
        count = 0;
        buffer = (Item[])new Object[size];
    }

    public synchronized void put(Item item) throws InterruptedException {...}

    public synchronized Item get() throws InterruptedException {...}

    public synchronized boolean isEmpty(){
        return count == 0;
    }

    public synchronized boolean isFull(){
        return count == buffer.length;
    }
}
```

PUT AND GET OPERATIONS

```
...
public synchronized void put(Item item) throws InterruptedException {
    while (isFull()){
        wait();
    }
    last = (last + 1) % buffer.length;
    count++;
    buffer[last] = item;
    notifyAll();
}

public synchronized Item get() throws InterruptedException {
    while (isEmpty()){
        wait();
    }
    first = (first + 1) % buffer.length;
    count--;
    notifyAll();
    return buffer[first];
}
...
```

- Is it really necessary to use `notifyAll`?

SECOND APPROACH FOR IMPLEMENTING MONITORS

- Exploiting explicit locks with **`ReentrantLock`** and **`Condition`** classes implementing condition variables provided by `java.util.concurrent` library
  - `ReentrantLock` class is a reentrant implementation of locks
    - analogous to mutexes (semaphores), but reentrant
  - `Condition` class represents condition variables to be used only inside blocks protected by a `ReentrantLock`
    - a condition object is strictly coupled to a specific reentrant lock
    - method to create a condition from a lock:
      - `public Condition` **`newCondition`**`();`
        - » returns a `Condition` instance for use with this `Lock` instance

IMPLEMENTING MONITORS WITH `ReentrantLock` AND `Condition`

- Implementing monitors with `ReentrantLock` and `Condition`:
  - `ReentrantLock mutex` for each monitor
  - wrapping each method with `mutex.lock` and `mutex.unlock`
  - for each condition to use, create it from the `mutex` lock
- In this case synchronized blocks / methods (intrinsic locks) are not used

EXAMPLE #2 REVISITED

- Synchronized cell

```
public class SynchCell2 {

  private int value;
  private boolean available;
  private Lock mutex;
  private Condition isAvail;

  public SynchCell2(){
    available = false;
    mutex = new ReentrantLock();
    isAvail = mutex.newCondition();
  }

  public void set(int v){
    try {
      mutex.lock();
      value = v;
      available = true;
      isAvail.signalAll();
    } finally {
      mutex.unlock();
    }
  }
  ...
```

EXAMPLE #2 REVISITED

- Synchronized cell

```
public class SynchCell {
  ...

  public int get() {
    try {
      mutex.lock();
       if (!available){
         try {
           isAvail.await();
         } catch (InterruptedException ex){}
      }
      return value;
    } finally {
      mutex.unlock();
    }
  }
}
```

- Note
  - methods are not synchronized
  - **`finally`** block, for ensuring mutex unlocking

BOUNDER BUFFER REVISITED (1/4)

```
public class BoundedBuffer<Item>  {
  private int first, last, count;
  private Item[] buffer;
  private Lock mutex;
  private Condition notFull, notEmpty;

  public BoundedBuffer(int size){
    first = last = count = 0;
    buffer = (Item[])new Object[size];
    mutex = new ReentrantLock();  // new ReentrantLock(true) for fair mutex
    notFull = mutex.newCondition();
    notEmpty = mutex.newCondition();
  }

  public void put(Item item) throws InterruptedException {...}
  public Item get() throws InterruptedException {...}
  public boolean isEmpty() throws InterruptedException {...}
  public boolean isFull() throws InterruptedException {...}
}
```

- Note
  - conditions are taken from the same lock

BOUNDER BUFFER REVISITED (2/4)

```
public class BoundedBuffer<Item>  {
  ...
  public boolean isEmpty() throws InterruptedException {
    try {
      mutex.lock();
      return count == 0;
    } finally {
      mutex.unlock();
    }
  }

  public boolean isFull(){
   try {
      mutex.lock();
      return count == buffer.length;
   } finally {
      mutex.unlock();
    }
  }
  ...
}
```

BOUNDER BUFFER REVISITED (3/4)

```
public class BoundedBuffer<Item>  {
  ...
  public void put(Item item) throws InterruptedException {
    try {
      mutex.lock();
      while (count == buffer.length){
        notFull.await();
      }
      last = (last + 1) % buffer.length;
      count++;
      buffer[last] = item;
      notEmpty.signal();
    } finally {
      mutex.unlock();
    }
  }
  ...
}
```

- Note
  - signaling the specific condition variable

BOUNDER BUFFER REVISITED (4/4)

```
public class BoundedBuffer<Item>  {
  ...
  public Item get() throws InterruptedException {
    try {
      mutex.lock();
      while (count == 0){
        notEmpty.await();
      }
      first = (first + 1) % buffer.length;
      count--;
      notFull.signal();
      return buffer[first];
   } finally {
      mutex.unlock();
    }
  }
  ...
}
```

SIGNALING SEMANTICS AND UNWANTED EFFECTS

- E = W < S
  - both in the case of implementations using `synchronized +wait/notify/notifyAll` and in the case of using `ReentrantLock+Condition`
- Effect (that could lead to unwanted behaviours...)
  - Given a thread T1 waiting on a condition C inside a monitor, a thread T2 which is inside the monitor, and a thread T3 waiting to enter the monitor
  - suppose that T2 signals the condition variable C
  - then T1 is awaked but since E = W, T1 competes for the monitor lock with T3
  - then, depending on the different implementations, *T3 could proceed and T1 wait.*

EXAMPLE

```
class MyMonitor {
   private ReentrantLock mutex;
  private Condition c;

  public MyMonitor(){
    mutex = new ReentrantLock();
    c = mutex.newCondition();
  }

  public  void m1() throws InterruptedException {
    try {
      mutex.lock();
      try {
        System.out.println("First thread inside,
                                        going to wait");
        c.await();
        System.out.println("First thread unblocked.");
        Thread.sleep(5000);
    } finally {
      mutex.unlock();
    }
  }

  ...
```

```
  ...

  public  void m2() throws InterruptedException {
    try {
      mutex.lock();
      System.out.println("Second thread inside");
       Thread.sleep(5000);
      System.out.println("Second thread inside,
                                    going to signal");
      c.signal();
      System.out.println("Second thread
                              inside signaled.");
    } finally {
      mutex.unlock();
    }
  }

  public  void m3() throws InterruptedException {
    try {
      mutex.lock();
      System.out.println("Third thread inside.");
      Thread.sleep(5000);
    } finally {
      mutex.unlock();
    }
  }
}
```

EXAMPLE

```
class MyThread1 extends Thread {
  private MyMonitor mon;

  public MyThread1(MyMonitor mon){
    this.mon = mon;
  }

  public void run(){
    log("First thread started.");
    mon.m1();
  }

  private void log(String msg){
    synchronized(System.out){
      System.out.println(msg);
    }
  }
}
```

```
class MyThread2 extends Thread {
  private MyMonitor mon;

  public MyThread2(MyMonitor mon){
    this.mon = mon;
  }

  public void run(){
    log("Second thread started.");
    mon.m2();
  }

  private void log(String msg){
    synchronized(System.out){
      System.out.println(msg);
    }
  }
}
```

```
class MyThread3 extends Thread {
  private MyMonitor mon;

  public MyThread3(MyMonitor mon){
    this.mon = mon;
  }

  public void run(){
    log("Third thread started.");
    mon.m3();
  }

  private void log(String msg){
    synchronized(System.out){
      System.out.println(msg);
    }
  }
}
```

```
public class TestSemantics {

  public static void main(String[] args) throws Exception {
    MyMonitor mon = new MyMonitor();
    new MyThread1(mon).start();
    new MyThread2(mon).start();
    new MyThread3(mon).start();
  }
}
```

EXAMPLE - OUTPUT

```
First thread started.
First thread inside, going to wait
Second thread started.
Second thread inside
Third thread started.
Second thread inside, going to signal
Second thread inside signaled.
Third thread inside.
First thread unblocked.
```

> The third thread enters *before* the first one